DURÉE: 2h, UNE FEUILLE A5 RECTO-VERSO MANUSCRITE AUTORISÉE

REMARQUES:
- Les exercices sont indépendants et peuvent être réalisés dans l'ordre voulu.
- Dans l'implémentation d'une méthode, on pourra utiliser n'importe quelle autre méthode définie auparavant même si celle-ci n'a pas été implémentée.
- Dans toutes les implémentations que vous écrivez, pensez à respecter le guide de syntaxe pour la programmation (règles de nommage, présentation des blocs, *etc.*).

EXERCICE 1: ALGORITHMIE (4 POINTS)

1. Écrire une fonction `str2Tab(chaine)` qui convertit une chaine de caractères en un tableau d'entiers.
   - On supposera que chaque nombre est séparé par une virgule.
   - On pourra utiliser la fonction `isnumeric()` des chaînes de caractères pour s'assurer qu'aucun mot ne s'est glissé dans la chaîne.
   - Par exemple, la chaîne "42,142,loutre,37,13,857" sera convertie en [42, 142, 37, 13, 857]

Solution: (1 point) : les solutions des questions sont "optimisées". On acceptera évidemment des solutions en plusieurs lignes.

```
def str2Tab( chaine ) :
  res = [ int(ssChaine) for ssChaine in chaine.split( "," ) if ssChaine.
     isnumeric( ) ]
  return res
```

2. Écrire une fonction `nbInf(tabInt, n)` qui renvoie le nombre d'entiers inférieur à n dans le tableau tabInt.

Solution: (1 point)

```
def nbInf( tabInt, n ) :
  res = sum( [ 1 for val in tabInt if val < n ] )
  return res
```

3. Écrire une fonction `isSup(tabInt, n)` qui renvoie la position du premier entier supérieur à n dans le tableau tabInt. Si aucun entier n'est supérieur à n, on renverra -1.

Solution: (1 point)

```
def isSup( tabInt, n ) :
  for i, val in enumerate( tabInt ) :
    if val >= n :      return i
  return -1
```

4. Écrire un programme principal qui :
   - déclare la chaîne de caractères "42,142,loutre,37,13,857",
   - la convertit en un tableau d'entiers tabInt et affiche le résultat,
   - demande à l'utilisateur un entier n1 puis affiche le nombre d'entiers inférieur n1 dans tabInt,
   - demande à l'utilisateur un entier n2 puis affiche la position du premier entier supérieur à n2 dans tabInt.

Solution: (1 point)

```
chaineInt = "42,142,loutre,37,13,857"
tabInt = str2Tab( chaineInt )
print( tabInt )
n1 = int( input( "Veuillez entrer un nombre: " ) )
print( nbInf( tabInt, n1 ) )
n2 = int( input( "Veuillez entrer un nombre: " ) )
print( isSup( tabInt, n2 ) )
```

EXERCICE 2: POLYMORPHISME ET APPELS DE FONCTIONS (5 POINTS)

On donne ci-dessous l'implémentation de 3 classes ainsi qu'un programme principal.

```
class Un :
  def __init__( self ) :  pass
  def m1( self, other ) :
    if isinstance( other, Un ) :          print( "1-1" )
    elif isinstance( other, Deux ) :      print( "1-2" )
    elif isinstance( other, Trois ) :     print( "1-3" )

class Deux (Un) :
  def __init__( self ) : super( ).__init__( )
  def m1( other, self ) :
    if isinstance( other, Deux ):         print( "2-2" )
    elif isinstance( other, Trois ):      print( "2-3" )
    elif isinstance( other, Un ):         print( "2-1" )

class Trois (Un) :
  def __init__( self ) : super( ).__init__( )
  def m1( self, other ) :
    if isinstance( other, Trois ) :       print( "3-3" )
    if isinstance( other, Deux ) :        print( "3-2" )
    elif isinstance( other, Un ) :        print( "3-1" )

un, deux, trois = Un( ), Deux( ), Trois( )
un.m1(un),       un.m1(deux),       un.m1(trois)
deux.m1(un),     deux.m1(deux),     deux.m1(trois)
trois.m1(un),    trois.m1(deux),    trois.m1(trois)
```

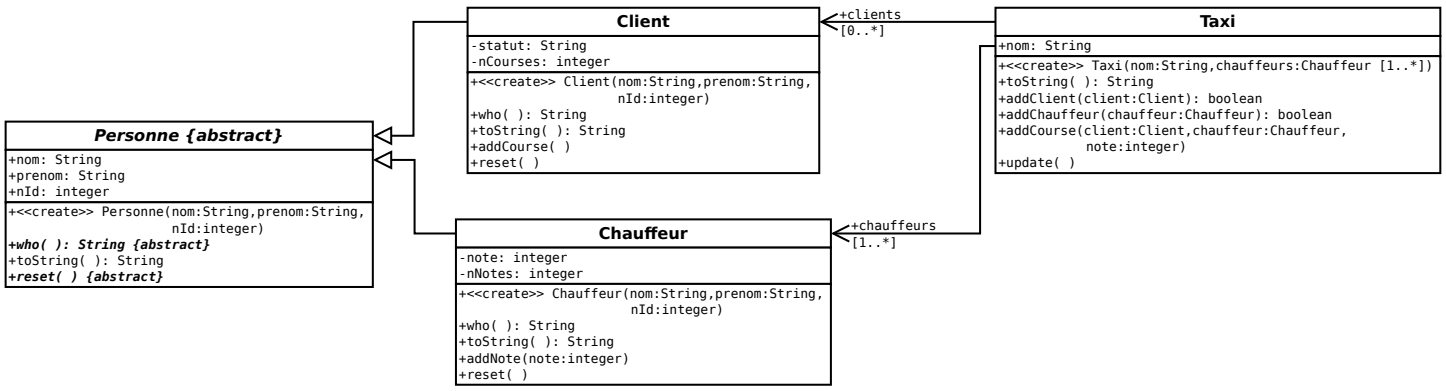
1. Sachant que le code s'exécute sans erreur, quels résultats sont affichés ?

Solution: (5 points) : 0.5 point par affichage (donc 1 point pour trois.m1(trois))

	Un	Deux	Trois	Remarques
`un.m1`	1-1	1-1	1-1	Un est la classe mère donc Deux et Trois sont des Un ⇒ les 2 clauses `elif` ne sont jamais testées
`deux.m1`	2-2	2-2	2-2	Inversion de `self` et `other` dans la déclaration de m1
`trois.m1`	3-1	3-2	3-3, 3-1	`if` au lieu de `elif` lors du test de Deux ⇒ 2 affichages pour un objet de classe Trois

Exercice 3: Programmation Orientée Objet (11 points)

On souhaite créer un logiciel de gestion de compagnie de taxis dont la modélisation UML est donnée ci-dessous. À tout moment, on pourra appeler n'importe quelle méthode présente sur le diagramme UML **même si celle-ci n'a pas encore été implémentée**. En revanche, les attributs privés n'auront **ni setters ni getters**.



1. **Classes** `Personne`, `Client` **et** `Chauffeur` : Une personne sera représentée par son nom, son prénom et un identifiant unique `nId`. La méthode `who( )` renverra le nom de la classe. Le statut d'un client dépendra du nombre de courses effectuées `nCourses` (Standard si moins de 5, Régulier entre 5 et 10, Premium si plus de 10). Le statut d'un client et la note d'un chauffeur pourront être réinitialiser par la méthode `reset`

(a) Implémenter la classe `Personne` (import, attributs, méthodes). On souhaite un affichage sous la forme "Prenom Nom (Classe, nID)".

Solution: (2 points) : 0.5 pour l'import, 0.5 pour les méthodes abstraites

```
from abc import ABC, abstractmethod
class Personne (ABC) :
  def __init__( self, nom, prenom, nId ) :
    self.nom = nom
    self.prenom = prenom
    self.nId = nId
  def __str__( self ) :
    res = self.prenom + " " + self.nom
    res += " (" + self.who( ) + ", " + str( self.nId ) + ")"
    return res
  @abstractmethod
  def who( self ) : pass
  @abstractmethod
  def reset( self ) : pass
```

(b) Pourquoi les méthodes `who` et `reset` sont elles abstraites ? Quelle(s) conséquence(s), cela a t'il sur la classe `Personne` ?

Solution: (1 point)
Ce sont 2 méthodes dont la classe `Personne` ne connaît pas le comportement (car elles seront implémentées dans les classes filles). Il faut donc les déclarer abstraites.
Comme la classe `Personne` possède des méthodes abstraites, la classe est abstraite (et ne peut pas être instanciée).

(c) Implémenter la classe `Client`. On souhaite un affichage sous la forme "Prenom Nom (Classe, nID) : nCourses, Statut".

Solution: (1,5 points) : 0.5 pour attributs privés, 0.5 pour l'utilisation de `super`

```
class Client (Personne) :
  def __init__( self, nom, prenom, nId ) :
    super( ).__init__( nom, prenom, nId )
    self.__nCourses = 0
    self.__statut = "Standard"
  def __str__( self ) :
    res = super( ).__str__( ) + ": " + str( self.__nCourses ) + ", " + self.
        __statut
    return res
  def who( self ) : return "Client"
  def addCourse( self ) :
    self.__nCourses += 1
    if 5 < self.__nCourses <= 10 :  self.__statut = "Regulier"
    elif 10 < self.__nCourses :     self.__statut = "Premium"
  def reset( self ) :
    self.__nCourses = 0
    self.__statut = "Standard"
```

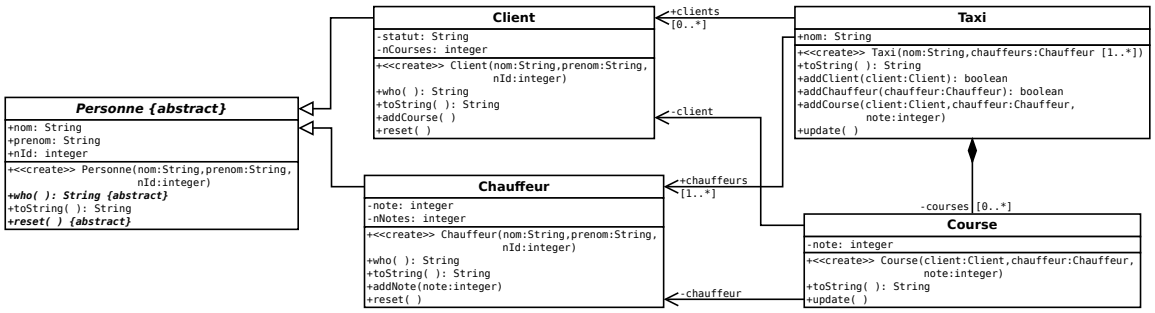
(d) Pourquoi les attributs `statut` et `nCourses` de la classe `Client` sont-ils privés et sans setters ?

Solution: (1 point)
Ces attributs sont privés car on ne souhaite pas qu'un client puisse modifier son statut et ainsi avoir des privilèges indus.

2. **Classe** `Course` : Une course sera modélisée par un client, un chauffeur et une note. Il faudra pouvoir l'afficher et lui permettre de mettre à jour les informations des clients (nCourses) et chauffeurs (note). De plus, une compagnie de `Taxi` conservera l'ensemble des courses effectuées.

(a) Recopier le diagramme UML sur votre copie et compléter le en y ajoutant la classe `Course`. Si une classe n'est pas modifiée, vous pourrez simplement la représenter par son nom (dans un rectangle).

Solution: (1 point)



(b) Justifier vos choix des visibilités des attributs de la classe Course ainsi que de ses relations avec les autres classes de l'application.

Solution: (1,5 points) : 0.5 par justification
Attributs privés pour éviter qu'un `Chauffeur / Client` ne s'attribue des courses qu'il n'a pas effectué ou modifie une note.
Association (unidirectionnelle) avec `Client` et `Chauffeur` car on a une simple utilisation de ces classes par `Course`.
Composition avec `Taxi`, car `Taxi` créé et ne partage pas ses `Courses` (une `Course` ne peut pas être effectuée par 2 compagnies différentes). On retrouve cette composition dans la méthode `addCourse` qui prend en entrée les informations nécessaires à la création d'une `Course` plutôt qu'une `Course` existante.

3. **Classe** `Taxi` :

(a) Implémenter la déclaration de la classe `Taxi` ainsi que le constructeur et la méthode `toString`.
On affichera le nom de la compagnie, le nombre de chauffeurs et de clients puis la liste des chauffeurs et clients (1 par ligne).

Solution: (1 point) : la fonction `__str__` est une suggestion. Compter juste dès lors que le cahier des charges est rempli

```
class Taxi :
  def __init__( self, nom, chauffeurs ) :
    self.nom = nom
    self.chauffeurs = chauffeurs
    self.clients = [ ]
    self.courses = [ ]

  def __str__( self ) :
    res = "La compagnie de taxis {0} a {1} chauffeur(s) et {2} client(s).\n"
        .format( self.nom, len( self.chauffeurs ), len( self.clients ) )
    res += "Les chauffeurs sont:\n  " + "\n  ".join( [ str(c) for c in self.
        chauffeurs ] ) + "\n"
    res += "Les clients sont:\n  " + "\n  ".join( [ str(c) for c in self.
        clients ] )
    return res
```

(b) Implémenter la méthode `addClient` qui ajoute un `Client` à l'attribut `clients` si celui-ci n'est pas déjà présent dans la liste. On affichera un message d'erreur compréhensible si l'ajout n'est pas possible et on renverra un booléen indiquant si l'ajout a réussi ou pas.

Solution: (1 point)

```
def addClient( self, client ) :
  if client not in self.clients :  # On s'assure que le client n'est pas
```

```
    self.clients.append( client )  # deja present dans la liste
    return True
  else :
    print( "Erreur: Client deja present" )
    return False
```

(c) Cette fonction nécessite t'elle de modifier la classe `Client` ou `Personne` ? Justifier votre réponse et donner l'implémentation éventuelle.

Solution: (1.5 points) : 0.5 pour la justification
Comme on vérifie que le Client n'existe pas déjà, il faut implémenter la fonction `__eq__` dans les classes Client et Personne.

```
# Dans la classe Personne
def __eq__( self, other ) :
  if not isinstance( other, Personne ) :  return False
  if self is other :                 return True
  # nId est un identifiant unique => pas besoin de comparer les
  if self.nId != other.nId :         return False    # autres attributs
  return True

# Dans la classe Client
def __eq__( self, other ) :
  """ On verifie qu'other est un Client (et pas un Chauffeur)
    puis comme nId est un identifiant unique, on peut simplement
    renvoyer le resultat du test de la classe mere """
  if not isinstance( other, Client ) :    return False
  return super( ).__eq__( other )
```

(d) Implémenter la méthode `update` qui réinitialise les statuts des clients et les notes des chauffeurs puis parcourt l'ensemble des courses pour les remettre à jour. On supprimera l'ensemble des courses à l'issue de cette étape.

Solution: (1 point)

```
def update( self ) :
  # On reinitialise les statuts des clients et notes des chauffeurs
  for c in self.clients :     c.reset( )
  for c in self.chauffeurs :  c.reset( )
  # On parcourt les courses pour calculer les nouveaux statuts et notes
  for c in self.courses :     c.update( )
  # On vide la liste de courses
  self.courses = [ ]
```